Small Tensor Operations on Advanced Architectures for High-order Applications

A. Abdelfattah[1], M. Baboulin[5], V. Dobrev[2], J. Dongarra[1,3], A. Haidar[1], I. Karlin[2], Tz. Kolev[2], I. Masliah[4], and S. Tomov[1]

[1] Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA
[2] Lawrence Livermore National Laboratory, Livermore, CA, USA
[3] University of Manchester, Manchester, UK
[4] Inria Bordeaux, France
[5] University of Paris-Sud, France

Abstract

This technical report describes our findings regarding performance optimizations of the tensor contraction kernels used in BLAST – a high-order FE hydrodynamics research code developed at LLNL – on various modern architectures. Our approach considers and shows ways to organize the contractions, their vectorization, data storage formats, read/write patterns, and parametrization as related to batched execution and parallelism in general. Autotuning framework is designed and used to find empirically best performing tensor kernels by exploring a large search space that results from the techniques described. We analyze these kernels to show the trade-offs between the various implementations, how different tensor kernel implementations perform on different architectures, and what tuning parameters can have a significant impact on performance. In all cases, we organize the tensor contractions to minimize their communications by considering index reordering that enables their execution as highly efficient batched matrix-matrix multiplications (GEMMs). We derive a performance model and bound for the maximum performance that can be achieved under the maximum data-reuse scenario, and show that our implementations achieve 90+% of these theoretically derived peaks on advanced multicore x86 CPU, ARM, GPU, and Xeon Phi architectures. These results significantly outperform what is available today in vendor libraries. In particular, we show average performance speedups of 1.3× and 2× compared to Intel MKL on two 10-core Haswell CPUs and KNL Xeon Phi, respectively, and 3× when compared to NVIDIA CUBLAS on the latest P100 NVIDIA GPU.

1 Introduction

Numerous important applications, and in particular high-order, tend to be cast in terms of many small matrix/tensor operations: they contain very large computations that consist of a large number of small tensors, which cannot be executed efficiently on accelerated platforms except in large groups, or batches. The emergence of large-scale, heterogeneous systems with GPUs and coprocessors has revealed the near total absence of linear algebra software for performing such small operations in batches.

The main challenge is that using the full potential of a processor is becoming increasingly difficult for small problems as the number of cores keeps increasing, along with increases in vector register sizes, while overheads for scheduling tasks and lower-level function calls are not decreasing fast enough to stay negligible compared to the computation times associated with these small problems. The complexity in the architectures of interest – multicore x86 CPUs, manycore Intel Xeon Phis, ARM, and NVIDIA GPUs – along with their memory hierarchies, for which we want to design efficient blocking strategies for small tensor contractions, is illustrated on Figure 1. Calling a sequence of small GEMMs through the standard BLAS API on a

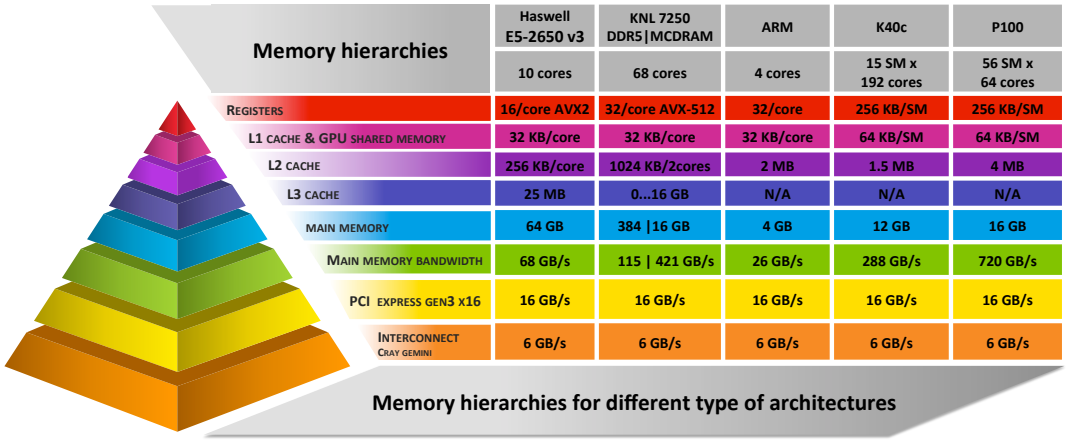

Figure 1: Memory hierarchies of the experimental CPU and GPU hardware

K40c GPU with 14 multiprocessors, for example, would result in performance that is at least 14× lower than expected. A solution that has led to success so far is to provide routines using a new batched BLAS API that exploit the extra level of parallelism while minimizing function call overheads. Although highly successful, this approach requires the development and optimization of various batched routines, where high-level algorithms and low-level kernels must be redesigned for full efficiency.

This was done in the recently released MAGMA Batched library, starting with the 1.6.1 release [20], to targeted small matrix factorizations (like LU, QR, and Cholesky) and a number of BLAS routines. For "small" sizes, in the range of 100 to 300, the MAGMA factorizations outperform similar routines in CUBLAS by 3 to 4× due to architecture- and size-aware algorithmic redesigns and optimizations. This report describes algorithms and optimization techniques that target smaller problem sizes, and in particular ones that are sub-warp/vector in size, and thus requiring vectorization across tensor operations.

2 Current development in tensors computations

We studied current tensor computations development to identify possible collaborations and ideas to explore. We found that the survey [17], and the references cited there, provide a good overview of the linear algebra aspects of tensors research, including higher-order tensor decompositions, their applications, and available software. A general overview with current and future direction in tensor research is presented in the *Future directions in tensor-based computation and modeling* workshop report [1].

There has been a number of software packages developed for tensors. Some are designed to be used through numerical computing mathematical environments like the Tensor Toolbox[1] for Matlab, GRTensor II[2] for Maple, or Ricci[3] for Mathematica, and therefore do not target accelerators, and high-performance computing in general. A few are specialized for particular applications, most notably for quantum chemical computations. For example, [19] presents

[1] http://www.sandia.gov/~tgkolda/TensorToolbox/
[2] http://grtensor.phy.queensu.ca/
[3] http://www.math.washington.edu/~lee/Ricci/

early work on using accelerators to accelerate tensor contractions on GPUs. The approach uses code generation techniques and is incorporated in the NW Chem computational chemistry suite. More recent work [25] also uses code generation techniques and autotuning (with a system called Baracuda, based on CUDA-CHiLL and a high-level module Optimizing Compiler with Tensor OPeration Intelligence (OCTOPI)) to report significant acceleration compared to NW Chem on particular tensor contractions.

Most approaches recognize and exploit the fact that many tensor contractions can be revamped into GEMMs. For example, [30] executes on CPUs contractions via GEMM on a properly ordered and structured tensor. As in the other approaches in quantum chemistry, large tensor contractions are targeted, although there are cases of small ones as well [32]. C. Cecka et al. [27] target a wide range of general tensor contractions on CPU and GPUs. The approach is library-based, investigating various index reordering leading to GEMMs, while trying to avoid memory movement. Focus is on small and moderate tensor sizes. Paul Springer and Paolo Bientinesi [31] also target general tensor contractions. They have a code generator that translates contractions to loop over GEMM code that can be applied to either large (compute-bound) or small (bandwidth-bound) tensor contractions. Performance is derived from GEMM implementations offered by tuned BLAS libraries.

In contrast to quantum chemistry, we target many but small contractions, that are often sub-warp/vector in size. In these cases, we have shown that tensor reshuffle operations to cast the contractions to GEMMs are also highly effective [5]. Moreover, we use a batched approach with custom-built BLAS kernels that are used to efficiently execute the tensor contractions [5, 6, 22]. Closely related in terms of general approach and applications targeted are the efforts of our collaborators P. Fischer et al. [21, 28], as well as P. Fisher, M. Min et al. [12, 26] who explored the use of OpenACC and CUDA Fortran for Nekbone – one of the core kernels of the incompressible Navier-Stokes solver Nek5000 – and showed 2.5× speedup of hybrid GPU+CPU computation over CPU-only performance on the same number of nodes (262,144 MPI ranks) on the Titan Cray XK7 supercomputer (nodes have one AMD Opteron 6274 16-core CPU and one Nvidia Tesla K20X GPU). Further, D. Medina, A. St-Cyr, and T. Warburton developed OCCA [23], a novel single kernel language that expands to OpenMP, OpenCL, and CUDA. OCCA is shown to deliver portable high-performance on different architectures and platforms (for finite difference, spectral element, and discontinuous Galerkin methods).

Hardware vendors so far do not provide software for tensor contractions. However, their optimized numerical libraries, and in particular BLAS, e.g., Intel's MKL and NVIDIA's CUBLAS, have been heavily used as the backend of the tensor contraction engines mentioned. Efforts to optimize for small matrix sizes have been done in the context of batched operations, or separate efforts, like Intel's libxsmm [16] or Nvidia's C. Cecka tensor contractions work [27].

Symmetry, when available, is also very beneficial to exploit. For example, E. Solomonik and J. Demmel have presented a new symmetry preserving algorithm that uses an algebraic reorganization in order to exploit considerably more symmetry in the computation of the contraction than the conventional approach, requiring fewer multiplications but more additions per multiplication [29].

3 Tensor operations in high order FEM

Here we derive the tensor formulations for high-order FE kernels. In particular, we are interested in formulations for the Lagrangian phase of the BLAST code developed at LLNL, which solves the Euler equations of compressible hydrodynamics in a moving Lagrangian frame [9, 10]. On

a semi-discrete level, the conservation laws of Lagrangian hydrodynamics can be written as:

$$\textbf{Momentum Conservation:} \quad \mathbf{M}_{\mathcal{V}} \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} = -\mathbf{F}\cdot\mathbf{1}, \tag{1}$$

$$\textbf{Energy Conservation:} \quad \frac{\mathrm{d}\mathbf{e}}{\mathrm{d}t} = \mathbf{M}_{\mathcal{E}}^{-1}\mathbf{F}^{T}\cdot\mathbf{v}\,, \tag{2}$$

$$\textbf{Equation of Motion:} \quad \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = \mathbf{v}, \quad \text{where} \tag{3}$$

$\mathbf{v}$, $\mathbf{e}$, and $\mathbf{x}$ are the unknown velocity, specific internal energy, and grid position, respectively; $\mathbf{M}_{\mathcal{V}}$ and $\mathbf{M}_{\mathcal{E}}$ are independent of time velocity and energy mass matrices; and $\mathbf{F}$ is the generalized corner force matrix depending on ($\mathbf{v}$, $\mathbf{e}$, $\mathbf{x}$) that needs to be evaluated at every time step.

BLAST improves the accuracy of simulations and provides a viable path to extreme parallel computing and exascale architectures. A core requirement to enable this path is the development of advanced methods and software to accelerate tensor assembly and evaluations on GPU and multi/many-core architectures. These operations exceed half of the BLAST simulation time.

To illustrate the tensor formulation of this problem, consider the computation of the finite element (FE) mass matrix M_E for an element (zone) E with a weight ρ. As a 2-dimensional tensor, M_E is:

$$(M_E)_{ij} = \sum_{k=1}^{nq} \alpha_k\, \rho(q_k)\, \varphi_i(q_k)\, \varphi_j(q_k)\, |J_E(q_k)|\,, \qquad i,j = 1,\ldots,nd, \quad \text{where}$$

nd is the number of degrees of freedom (dofs), nq is the number of quadrature points, $\{\varphi_i\}_{i=1}^{nd}$ are the FE basis functions on the reference element, $|J_E|$ is the determinant of the element transformation, and $\{q_k\}_{k=1}^{nq}$ and $\{\alpha_k\}_{k=1}^{nq}$ are the points and weights of the quadrature rule.

Tensors can be introduced by taking the $nq \times nd$ matrix B, $B_{ki} = \varphi_i(q_k)$, and the $nq \times nq$ diagonal matrix D_E, $(D_E)_{kk} = \alpha_k\, \rho(q_k)\, |J_E(q_k)|$. Then, $(M_E)_{ij} = \sum_{k=1}^{nq} B_{ki}(D_E)_{kk}B_{kj}$, i.e.,

$$M = B^T D B \;\; \text{(omitting the element index } E).$$

Using FEs of order p with a quadrature rule of order p, we have $nd = O(p^d)$ and $nq = O(p^d)$, so B is a dense $O(p^d) \times O(p^d)$ matrix. If the FE basis and the quadrature rule have tensor product structure, then in 2D,

$$M_{i_1,i_2,j_1,j_2} = \sum_{k_1,k_2} (B^{1d}_{k_1,i_1} B^{1d}_{k_1,j_1})(B^{1d}_{k_2,i_2} B^{1d}_{k_2,j_2}) D_{k_1,k_2}\,, \tag{4}$$

where B^{1d} is a dense $O(p) \times O(p)$ matrix and D is a dense $O(p) \times O(p)$ matrix. In 3D,

$$M_{i_1,i_2,i_3,j_1,j_2,j_3} = \sum_{k_1,k_2,k_3} (B^{1d}_{k_1,i_1} B^{1d}_{k_1,j_1})(B^{1d}_{k_2,i_2} B^{1d}_{k_2,j_2})(B^{1d}_{k_3,i_3} B^{1d}_{k_3,j_3}) D_{k_1,k_2,k_3}, \quad \text{where} \tag{5}$$

D is a dense $O(p) \times O(p) \times O(p)$ tensor, and $i = (i_1, \cdots, i_d)$, $j = (j_1, \cdots, j_d)$, $k = (k_1, \cdots, k_d)$ are the decompositions of the dof and quadrature point indices into the tensor components along logical coordinate axes. Note that if we store the tensors as column-wise 1D arrays, then

$$M^{nd_1 \times nd_2 \times nd_1 \times nd_2}_{i_1,i_2,j_1,j_2} = M^{nd \times nd}_{i,j} = M^{nd^2}_{i+nd\,j} = M^{nd^2}_{i_1+nd_1 i_2+nd(j_1+nd_1 j_2)}\,,$$

i.e. we can reinterpret M as a $nd \times nd$ matrix, or a fourth order tensor of dimensions $nd_1 \times nd_2 \times nd_1 \times nd_2$, or a vector of dimension nd^2, without changing the underlying storage.

More generally, given a $n_1 \times \cdots \times n_r$ tensor T, we can reshape it as a $m_1 \times \cdots \times m_q$ tensor

$$Reshape(T)^{m_1 \times \cdots \times m_q}_{j_1,\cdots,j_q} = T^{n_1 \times \cdots \times n_r}_{i_1,\cdots,i_r}$$

as long as $n_1 n_2 \cdots n_r = m_1 m_2 \cdots m_q$ and

$$i_1 + n_1 i_2 + \cdots + n_1 n_2 \cdots n_{r-1} i_r = j_1 + m_1 j_2 + \cdots + m_1 m_2 \cdots m_{q-1} j_q$$

for $i_k = 0, \ldots, n_k - 1$, $k = 1, \ldots, r$ and $j_l = 0, \ldots, m_l - 1$, $l = 1, \ldots, q$. In particular, we can reduce the number of dimensions of the tensor by agglomerating consecutive indices as above, and we can also increase the number of dimensions by simply specifying the size in the artificial dimensions to be one:

$$Reshape(T)^{1 \times n_1 \times \cdots \times n_r}_{0,i_1,\cdots,i_p} = T^{n_1 \times \cdots \times n_r}_{i_1,\cdots,i_r} = Reshape(T)^{n_1 \times \cdots \times n_r \times 1}_{i_1,\cdots,i_p,0}.$$

The action of the operator, $U = MV$, can be written in the tensor product case as

$$U_{i_1,i_2} = \sum_{k_1,k_2,j_1,j_2} B^{1d}_{k_1,i_1} B^{1d}_{k_2,i_2} D_{k_1,k_2} B^{1d}_{k_1,j_1} B^{1d}_{k_2,j_2} V_{j_1,j_2}, \text{ and} \tag{6}$$

$$U_{i_1,i_2,i_3} = \sum_{k_1,k_2,k_3,j_1,j_2,j_3} B^{1d}_{k_1,i_1} B^{1d}_{k_2,i_2} B^{1d}_{k_3,i_3} D_{k_1,k_2,k_3} B^{1d}_{k_1,j_1} B^{1d}_{k_2,j_2} B^{1d}_{k_3,j_3} V_{j_1,j_2,j_3}\,. \tag{7}$$

Given B^{1d}, D, and V, tensors M and U must be computed based on the generalized contractions (4)–(7). The matrix sizes are relatively small, e.g., with $p = 2..8$ and $d = 2$ or 3. The computations can be parallelized between the elements, and we may need to perform multiple evaluations with (3)–(4).

Remark 1. *One natural way to implement the above contractions is as a sequence of pairwise contractions, i.e. by evaluating the sums one at a time. While this is an efficient approach, there is enough complexity here that maybe one can come up with something better?*

The table below summarizes the complexity and storage of the general and nearly-optimal assembly and evaluation algorithms, and the pairwise tensor kernel operations that we need for the (1)–(4) operations above. Some of these kernels can be reduced further to a common kernel. For example the contractions (3a)–(4f) can be implemented as tensor index reordering (generalized transpose) plus the dgemm $A, B \mapsto A^T B$. This is because contraction by the first index, for example (4f), can be written as $Reshape(A)^{nd_1 \times (nd_2 nd_3)} = B^{1d}\ Reshape(C)^{nq_1 \times (nd_2 nd_3)}$. (If the contraction is not by the first index, reducing it to $A^T B$ requires data reordering and it will be interesting to explore when and if that is worth it or not.) Without reordering, all of the (3a)–(4f) operations can be implemented through reshaping and the kernels

$$A_{i,j,k,l} = \sum_s B_{i,s,j} C_{k,s,l} \qquad \text{and} \qquad A_{i,k,l,j} = \sum_s B_{i,s,j} C_{k,s,l}\,. \tag{8}$$

For example, the matrix multiplication (3b) of two $m \times p$ and $p \times n$ matrices B and C can be implemented by reshaping B to $m \times p \times 1$, C to $1 \times p \times n$, applying the $A_{i,j,k,l}$ kernel above, and reshaping the $m \times 1 \times 1 \times n$ result to a $m \times n$ matrix.

The matrix assembly contractions (1a)–(2c) can also be reduced to a common kernel (plus reshaping):

$$A_{k,i,l,j} = \sum_s B_{s,i} B_{s,j} C_{k,s,l}. \tag{9}$$

For example, in (2c) we can first reshape C as a third order tensor by agglomerating $i1, i2$ and $j1, j2$, apply the above kernel, and reshape the result as a 6th order tensor (or a $nd \times nd$ matrix).

stored items	assembly FLOPs	storage amount	matvec FLOPs	numerical kernels	
full assembly					
M	$O(p^{3d})$	$O(p^{2d})$	$O(p^{2d})$	$B, D \mapsto B^T D B$, $x \mapsto Mx$	
decomposed evaluation					
B, D	$O(p^{2d})$	$O(p^{2d})$	$O(p^{2d})$	$x \mapsto Bx$, $x \mapsto B^T x$, $x \mapsto Dx$	
near-optimal assembly – equations (1) and (2)					
$M_{i_1,\cdots,j_d}$	$O(p^{2d+1})$	$O(p^{2d})$	$O(p^{2d})$	$A_{i_1,k_2,j_1} = \sum_{k_1} B^{1d}_{k_1,i_1} B^{1d}_{k_1,j_1} D_{k_1,k_2}$	(1a)
				$A_{i_1,i_2,,j_1,j_2} = \sum_{k_2} B^{1d}_{k_2,i_2} B^{1d}_{k_2,j_2} C_{i_1,k_2,j_1}$	(1b)
				$A_{i_1,k_2,k_3,j_1} = \sum_{k_1} B^{1d}_{k_1,i_1} B^{1d}_{k_1,j_1} D_{k_1,k_2,k_3}$	(2a)
				$A_{i_1,i_2,k_3,j_1,j_2} = \sum_{k_2} B^{1d}_{k_2,i_2} B^{1d}_{k_2,j_2} C_{i_1,k_2,k_3,j_1}$	(2b)
				$A_{i_1,i_2,i_3,j_1,j_2,j_3} = \sum_{k_3} B^{1d}_{k_3,i_3} B^{1d}_{k_3,j_3} C_{i_1,i_2,k_3,j_1,j_2}$	(2c)
near-optimal evaluation (partial assembly) – equations (3) and (4)					
B^{1d}, D	$O(p^d)$	$O(p^d)$	$O(p^{d+1})$	$A_{j_1,k_2} = \sum_{j_2} B^{1d}_{k_2,j_2} V_{j_1,j_2}$	(3a)
				$A_{k_1,k_2} = \sum_{j_1} B^{1d}_{k_1,j_1} C_{j_1,k_2}$	(3b)
				$A_{k_1,i_2} = \sum_{k_2} B^{1d}_{k_2,i_2} C_{k_1,k_2}$	(3c)
				$A_{i_1,i_2} = \sum_{k_1} B^{1d}_{k_1,i_1} C_{k_1,i_2}$	(3d)
				$A_{j_1,j_2,k_3} = \sum_{j_3} B^{1d}_{k_3,j_3} V_{j_1,j_2,j_3}$	(4a)
				$A_{j_1,k_2,k_3} = \sum_{j_2} B^{1d}_{k_2,j_2} C_{j_1,j_2,k_3}$	(4b)
				$A_{k_1,k_2,k_3} = \sum_{j_1} B^{1d}_{k_1,j_1} C_{j_1,k_2,k_3}$	(4c)
				$A_{k_1,k_2,i_3} = \sum_{k_3} B^{1d}_{k_3,i_3} C_{k_1,k_2,k_3}$	(4d)
				$A_{k_1,i_2,i_3} = \sum_{k_2} B^{1d}_{k_2,i_2} C_{k_1,k_2,i_3}$	(4e)
				$A_{i_1,i_2,i_3} = \sum_{k_1} B^{1d}_{k_1,i_1} C_{k_1,i_2,i_3}$	(4f)
matrix-free evaluation					
none	none	none	$O(p^{d+1})$	evaluating entries of B^{1d}, D, (3a)–(4f) sums	

Table 1: Main tensor contractions needed in high-order FEMs

4 Main activities

The main focus of our developments was the optimization of the tensor contractions in BLAST, as summarized in Table 1. These contractions are expressed as batched GEMMs of very small sizes. The approach is described in Section 3. Note that we can restrict the contractions needed to three main cases. Therefore, we can afford to directly target these cases for optimization, which is in contrast to the general approaches that rely on a generator to express a general tensors contraction to GEMMs (see Section 2). We can consider such generator in an extension of the current developments to a general tensor contractions library.

We investigated various techniques to develop high-performance batched GEMMs of small sizes on various architectures. The sizes were taken to be up to 32 and the target architectures included multicore x86 CPUs, ARM, GPUs, and Intel Xeon Phis.

The overall design and coding approaches that we took are described in Section 5. Algorithms, optimization strategies, and issues are given in Section 6. Performance results are presented in Section 7. All these developments were presented and published at a number of conferences:

- **Smoky Mountains Computational Sciences and Engineering Conference** (SMC'15):

we presented a poster giving an overview and our plans towards developing a high-performance tensor algebra package for accelerators [7];

- **Smoky Mountains Computational Sciences and Engineering Conference** (SMC'16): we presented a poster on tensor contractions for high-order FEM on CPUs, GPUs, and KNLs [13];
- **The International Conference on Computational Science (ICCS'16)**: paper on high-performance tensor contractions for GPUs [5];
- **ISC High Performance (ISC 2016)**: paper on batched GEMM for GPUs [6];
- **22Nd International Conference on Euro-Par 2016**: paper on small GEMMs for CPU and GPU architectures [22];
- **Workshop on Batched, Reproducible, and Reduced Precision BLAS**: ICL organized two workshops in the past year with the intent to extend the BLAS standard [2, 3, 11] to include batched computations (and others). We presented our work on tensor contractions;
- **SIAM CSE'17**: we presented our work on tensor contractions [4].

A. Haidar and S. Tomov visited LLNL in 2016 and presented the current project status and work on tensor contractions.

5 Tensor contraction interfaces and code generation

To develop high-quality HPC software for tensor contractions, we impose the following three main requirements on our interface design:

Convenience of use: Interfaces of HPC libraries are extremely important for the libraries' adoption by the community. Interfaces must provide convenience and practicality of use, including ease of interoperability between libraries. Ideally, a tensor data type standard must be defined. The standard for a dense matrix is a pointer, sizes, leading dimension, and assumption for column-major data layout, e.g., as in BLAS and LAPACK. For tensors, motivated by reviewing interfaces in available libraries and the success of BLAS, we propose to represent a tensor by its dimensions and a data layout abstraction. The abstraction is to provide a choice of predefined layouts, and ways to switch it, or to easily add user-specified layouts, without changing the underlying algorithms. In general, a specific layout provides the formula of mapping the multi-dimensional tensor to the memory, e.g., a second order tensor can be stored as a column-major matrix A with leading dimension lda, in which case the abstraction maps $A_{i,j}$ to $A[i + j * lda]$ (see Listing 1).

Readability: Numerical software must be understandable, which is needed for ease of maintenance, as well as code optimizations, and testing. While we can easily implement any interface, e.g., even expressing the interface and tensor APIs in a DSEL if needed (plus code generation techniques to translate the DSEL to a standard language; see the example in Einstein tensor notations on Figure 2), we determined that it is better to provide implementations relying on a standard language and the code generation features provided within the language. The C++14 standard for example is expressive enough to allow us to implement readable and easy to use interfaces.

Performance: While we expect to obtain high performance mostly through algorithmic design and autotuning (see Section 6), we do not want to compromise on optimization opportunities like removing parameters checking and loop unrolling to eliminate jumps and loop

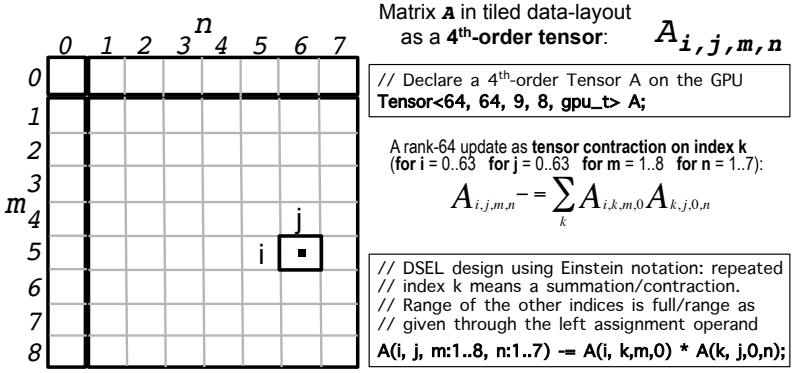

Figure 2: Example of a 4^{th}-order tensor resulting from tile matrix layout used in dense LA, a tensor contraction, and a possible tensor contractions design using Einstein summation notation and a Domain Specific Embedded Language (or *DSEL*).

count decrements. These optimizations are essential especially for the small computations that we target. Therefore, in our design we consider the use of compiler features related to code generation (e.g., templates, etc.), as further discussed below.

Related to performance, a lost optimization opportunity is if static checking and compile time information is not provided as part of the scientific libraries used. As an example, LAPACK routines start by checking entry parameters dynamically, which results in extra time for small size computations. The type of algorithms that we intend to use, e.g., as the MAGMA Batched algorithms, also require specific tuning [14] as performance will greatly vary depending on numerous parameters. To avoid these performance drawbacks, and also benefit from optimization techniques like compiler loop unrolling for static dimension problems, we use features of the new C++14 standard. In particular, the `constexpr` specifier enables to evaluate the value of a function or variable at compile time. We use this feature with integral constants and template specialization to design our tensor dimensions layout:

```
// Template specialization
constexpr auto layout = of_size<5,3>();
// Using Integral constant
constexpr auto layout1 = of_size(5_c,3_c);
// Using dynamic dimensions
constexpr auto layout2 = of_size(5,3);
// Access Dimensions at compile time
constexpr auto dim1 = layout(1);
```

Listing 1: Dimensions for Tensors

```
// Create a rank 2 tensor of size 5,3 on GPU
constexpr tensor<float,gpu_> d_ts(of_size<5,3>());
// Create a rank 2 tensor of size 5,3 on CPU
constexpr tensor<float,gpu_> d_ts(of_size<5,3>());
// Create a rank 2 tensor of size 5,3 on CPU
constexpr tensor<float> ts(of_size<5,3>());
// Use thrust to fill d_ts with 9
thrust::fill(d_ts.begin() , d_ts.end() , 9);
// Copy d_ts from GPU to ts on CPU
copy( d_ts , ts ) ;
```

Listing 2: Create Tensor and copy

As seen in Listing 1, we propose 3 ways to specify dimensions using the function *of_size* which returns a layout containing the static or dynamic dimension. Each operator inside the layout uses the `constexpr` keyword which enables us to return sizes at compile time if possible.

We can then freely extract the dimensions (Listing 1) and use them to specify our CPU and GPU kernels at compile time. Our tensor model is based on our layout, the data type of the tensor and its locality. The memory buffer is based on vector from the Standard Template Library for CPU and thrust [8] for GPU. To generate a tensor, we need to pass a data type and locality as template parameter and the size to the constructor (Listing 2).

Transfers between CPU and GPU tensors can be expressed through the function copy (Listing 2). This function will use the stream 0 by default but a stream can be passed for asyn-

chronous copy. We have designed two models for batched computing (Listing 3). The first one is based on allocating a single memory block for all tensors to improve data transfers and locality while the other is a group of same size tensors.

```
// Create a batch that will contain 15 tensors of size 5,3,6
constexpr auto batch<float, gpu_> b = make_batch(of_size(5_c,3_c,6_c) , 15);
// Accessing a tensor from the batch returns a view on it
constexpr auto view_b = b(0);
// Create a grouping of tensors of same size tensors
constexpr auto group<float,gpu_> g(of_size(5_c,3_c));
// Add a tensor to the group
constexpr auto tensor<float,gpu_> d_ts( of_size(5_c,3_c) );
g.push_back(d_ts);
```

Listing 3: Batched tensors

Once we have defined these functions we can call the kernel to compute a batched dgemm on tensors of dimension 2.

```
constexpr auto batch<float, gpu_> b =  make_batch(of_size(5_c,3_c) , 15);
constexpr auto batch<float, gpu_> b1 = make_batch(of_size(5_c,3_c) , 15);
// Product of two tensor batched of dimension 2 for matrix product using C++ operator
constexpr auto c = b * b1;
// Product using the batch dgemm function that can be specialized depending on parameters
constexpr auto c = batch_gemm(b , b1 );
```

Listing 4: Tensor Operations

Using the above features of the C++14 standard, we also overloaded the functions dealing with the different SIMD instructions on different architectures, e.g., Intel SSE, AVX2, AVX512, and ARM AArch64. This allows us to easily support different SIMD extensions while using a generic function for each call. These programming techniques allow us to have a single source file that supports Intel and ARM processors for very efficient small size matrix products. It is also very simple to extend. For example, adding support for IBM processors with Altivec SIMD instructions only requires us to add an overload for each SIMD functions we need.

6 Algorithms design for performance and portability

The generalized tensor contractions (4)–(7) can be summarized to a few kernels [7] (as discussed in Section 3). One natural way to implement them is as a sequence of pairwise contractions, i.e. by evaluating the sums one at a time. While this is an efficient approach, especially when coupled with a batched approach as in MAGMA [14, 15], there is enough complexity here that maybe one can come up with something better. Indeed, a number of the kernels needed can be reduced further to a tensor index reordering (generalized transpose) plus the dgemm $A, B \mapsto A^T B$. This is because contraction by the first index, for example

$$C_{i_1,i_2,i_3} = \sum_{k_1} A_{k_1,i_1} B_{k_1,i_2,i_3},$$

can be written as $Reshape(C)^{nd_1 \times (nd_2 nd_3)} = A^T \; Reshape(B)^{nq_1 \times (nd_2 nd_3)}$. If the contraction is not by the first index, reducing it to $A^T B$ requires data reordering. However, we note that there is a way not to do this explicitly since the matrices are very small; we organize our algorithms (next) to have a phase of reading A and B to shared memory, followed by a computational phase. Thus, the reading can be from consecutive or not data, and in either case to benefit from data reuse (of the small A and B in shared memory) using the same computational phase.

In summary, all of the (6)–(7) operations can be implemented through reshaping and the kernels $A_{i,j,k,l} = \sum_s B_{i,s,j} C_{k,s,l}$ and $A_{i,k,l,j} = \sum_s B_{i,s,j} C_{k,s,l}$.

The matrix assembly contractions (4)–(5) can also be reduced to a common kernel (plus reshaping): $A_{k,i,l,j} = \sum_s B_{s,i} B_{s,j} C_{k,s,l}$.

6.1 Performance model

To evaluate the efficiency of our algorithms we derive theoretical bounds for the maximum achievable performance $P_{max} = F/T_{min}$, where F is the flops needed and T_{min} is the fastest time to solution. For simplicity, consider $C = \alpha AB + \beta C$ on square matrices of size n. We have $F \approx 2n^3$ and $T_{min} = min_T(T_{Read(A,B,C)} + T_{Compute(C)} + T_{Write(C)})$. Note that we have to read/write $4n^2$ elements, or $32n^2$ Bytes for double precision (DP) calculations. Thus, if the maximum achievable bandwidth is B (in Bytes/second), we take $T_{min} = 4n^2/B$ in DP. Note that this time is theoretically achievable if the computation totally overlaps the data transfers and does not disrupt the maximum rate B of read/write to the GPU memory. Thus,

$$P_{max} = \frac{2n^3 B}{32n^2} = \frac{nB}{16} \text{ in DP.}$$

The achievable bandwidth can be obtained by benchmarks, e.g., using NVIDIA's `bandwidthTest`. For example, on a K40 GPU with ECC on the peak is 180 GB/s, so in that case P_{max} is 11.25 n GFlop/s (denoted by the diagonal lines on the performance Figures in Section 7). Thus, when $n = 16$ for example, we expect a theoretical maximum performance of 180 GFlop/s in DP.

6.2 Algorithms for tensor contractions through batched GEMMs

As described, we transform the tensor contractions to batched GEMM. To achieve HP on batched GEMM for small matrices that are sub-warp in size, we apply the following techniques:

- Using templates and `constexpr` specifiers we manage to avoid parameters checking and get compiler-unrolled loops in our kernels;
- We avoid passing arrays of pointers to the batched matrices, which are quite large, by passing them through formula/function that is part of the tensor's structure definition;
- The kernels are organized as follows: (1) Read A and B into shared memory; (2) Compute AB in registers; (3) Update C. Reading A, B, and C is through functions in the tensor's structure definition, which allows us to work with matrices that are not stored in the standard matrix format. Thus, we do not need explicit data reordering for some contractions, in order to efficiently benefit from the GEMM computation (step 2 above);
- We developed versions based on: how A and B are read; prefetching or not C by intermixing reading C with the computation in (2); number of matrices handled by a thread block, combined with prefetching variations for A and B; number of threads per thread block; and combinations of the above.

In all cases, to maximize data reuse, a single GEMM is done on a single thread block. Batching of the computation is done as in [15]. A GPU GEMMs technique, used for large matrices since the Fermi architecture [24], is to apply hierarchical communications/blocking on all memory levels, including a new register blocking. Register blocking can be added by making a single thread compute more than one element of a resulting matrix (in the same row or column, so that when an element from A or B is loaded into a register, it gets used more than once). Current results show that register blocking may not be needed in this case.

6.3 Autotuning

The complexity of tuning our algorithms is handled through autotuning [18, 6]. We note that although tuning is important, the algorithmic design (as in Section 6.2) is the more critical part

for obtaining high-performance, as the overall success is limited by the quality of the algorithms used. With that in mind, there are generally two kinds of approaches for doing autotuning – model-driven optimization and empirical optimization. We use a combination. The model-driven part comprises compiler code generation and optimizations (as in Section 5). For the empirical optimization, a large number of parametrized code variants are generated and run on a given platform to discover the one that gives the best performance. The effectiveness of empirical optimization depends on the chosen parameters to optimize, and the search heuristic used. Our GEMM gesign and some of its paramers are illustrated on Figure 3. The GPU

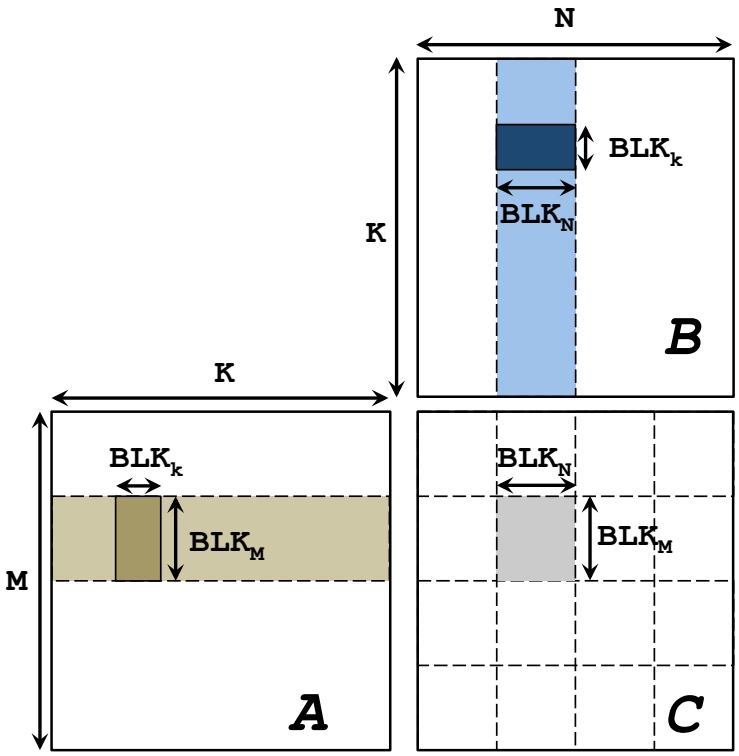

Figure 3: GEMM design and (auto)tuning parameters for various levels of blocking.

implementation, for example, uses register blocking to hold three blocks of A, B, and C in the register file. As a thread block (TB) moves across A and B, new blocks of A and B are read in the registers, while the C block is kept for result accumulation. The multiplication between blocks takes place in shared memory, which allows data prefetching of the next A and B blocks in registers. The kernel has at least five tuning parameters, which are $\texttt{BLK}_\texttt{M}$, $\texttt{BLK}_\texttt{N}$, $\texttt{BLK}_\texttt{K}$, and the (x, y) configuration of TBs.

The designs of these algorithms and their implementations using templates are very convenient in setting up our autotuning framework. The process is illustrated on Figure 4.

Further details on the autotuning framework can be found in [6], while details on the kernels, including their extension to multicore CPUs, are available in our technical report [22].

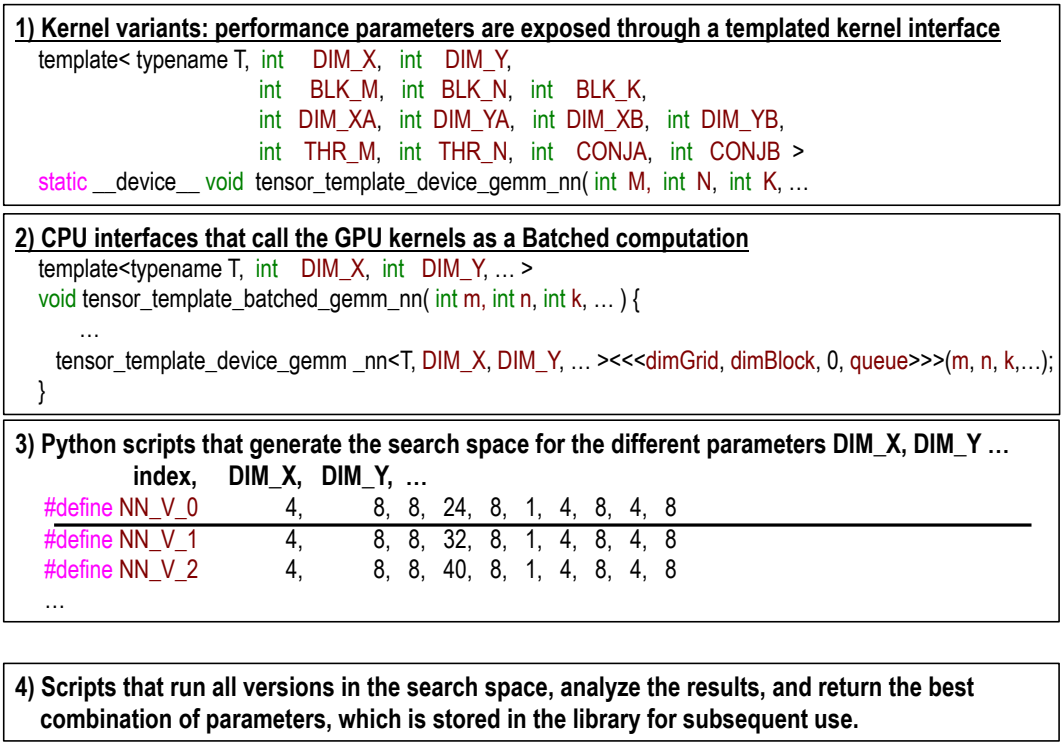

Figure 4: Autotuning framework for high-performance tensor contractions in MAGMA.

7 Experiments on tensor computations

We designed a framework and parametrized implementations, as described in Section 6, and autotuned them for the architectures of interest (shown in Figure 1). The performance results are presented in the following subsections for multicore CPUs, ARM, GPU, and Xeon Phi, respectively. We note that in all cases the goal was to reach close to the theoretical peak performance, as derived in Section 6.1.

We used gcc compiler 5.3.0 for our CPU code (with options -std=c++14 -O3 -avx -fma), as well as the icc compiler from the Intel suite 2016.0.109, and the BLAS implementation from MKL (Math Kernel Library) 16.0.0. We used CUDA Toolkit 8.0 for the GPU.

7.1 Multicore results

Figure 5 shows the performance of batched DGEMMs on a 10-cores Intel Xeon E5-2650 v3 (Haswell). The batch count is 10,000. For the MKL library we used the best of the following two: 1) An OpenMP loop statically or dynamically unrolled among the cores (we choose the best results), where each core computes one matrix-matrix product at a time using the optimized sequential MKL dgemm routine, or 2) The batched dgemm routine that has been recently added to the MKL library.

Note that both MAGMA and MKL achieve very good, close the the peak performance, with MAGMA outperforming MKL. For very small matrices, like the ones of interest, performing

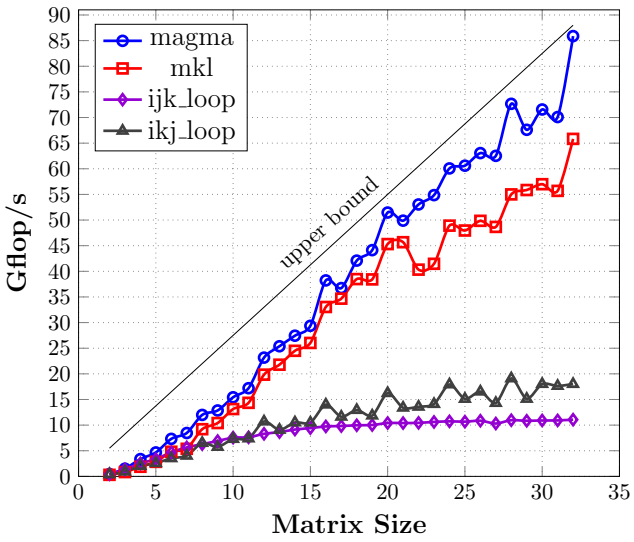

Figure 5: Performance comparison of batched DGEMMs on Haswell CPU.

proper register blocking is essential for getting good performance on CPUs (among the other techniques mentioned). In particular, once data is loaded into a SIMD register, it must be reused as much as possible before being replaced by new data. The Haswell CPU has 16 (AVX-2) SIMD registers, 256-bit each. Thus, to load completely an 8×8 matrix, for example, would take all 16 registers. Therefore, one can expect that loading the whole B will not be optimal as we will have to reload the vectors for A and C. However, if we load only 8 registers for B, which is equal to 4 rows, we can compute a row of C at each iteration and reuse these 8 registers for each iteration. All combinations like this are checked to find the best one using our autotuning framework. Also, note that the simple loop-based implementations can not be register-blocked and vectorized automatically by the compiler, which results in poor performance (although data is in L1 cache).

Multiple CPUs have non-uniform memory accesses based on the data location. Figure 6 illustrates that we can increase performance if we take NUMA effects into consideration. Note that we manage to double the performance from a single to two sockets, meaning we manage to double the bandwidth (44 GB/s for a socket). Interleaving the memory allocation between the sockets (by memory pages) helps, but better performance is achieved if we explicitly allocate half the matrices on one socket and the other half on the second socket (see the performance marked by `custom numa`).

7.2 ARM results

Figure 7 gives the performance that we obtain on an ARM processor – Tegra X1, a 4-core Cortex A57. We apply the same approach as for multicore CPUs. The difference is that the ARM intrinsics only support 128-bit vectors, which severely limit the SIMD use for double precision computations. However, the autotuning that we use manages to resolve this difference without extra efforts.

As in the other cases, we outperform other available libraries (OpenBLAS in this case) and we are very close to the theoretical peak. Simple loops again can not be vectorized and blocked

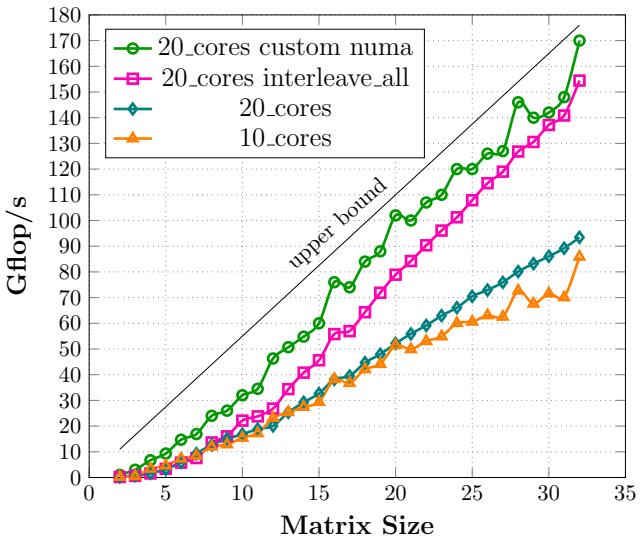

Figure 6: Effect of the NUMA memory management on multi-socket Haswell CPU.

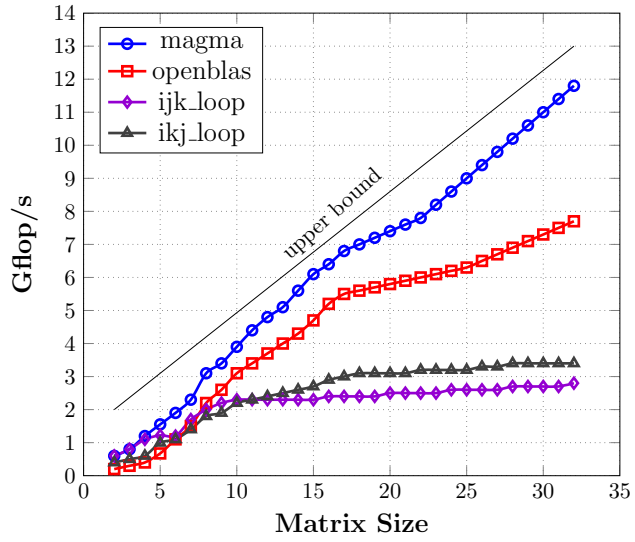

Figure 7: Experimental results of the matrix-matrix multiplication on the Tegra X1.

well for register reuse.

7.3 GPU results

Figure 8 shows the DP performance of four of our autotuned kernels from Section 6.2. Each of the kernels gives best results for certain dimensions. Shown is also the performance on 16 cores Intel Sandy Bridge CPUs. The implementation uses OpenMP to run in parallel a GEMM per thread, where the GEMMs call MKL. For $n = 16$ the speedup is about $4\times$ and grows for smaller sizes. Similar trend is observed in a comparison to the batched DGEMM in CUBLAS.

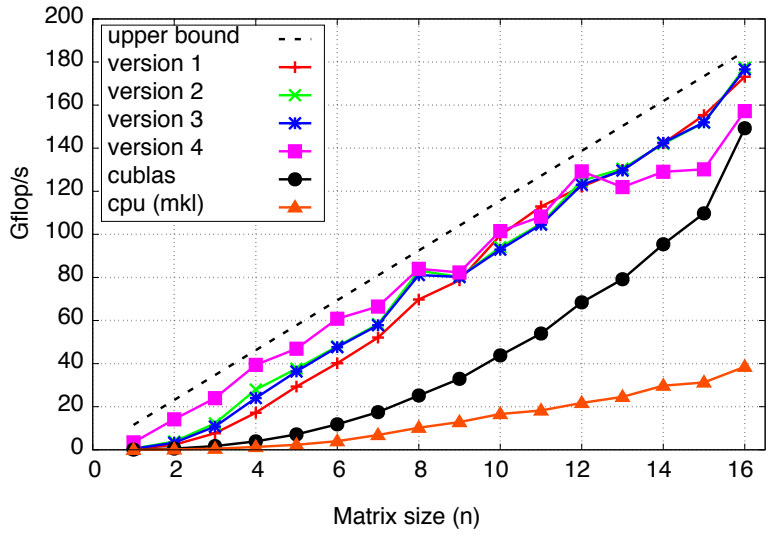

Figure 8: Performance comparison of tensor contraction versions using batched $C = \alpha AB + \beta C$ on 100,000 square matrices of size n (x-axis) on a K40c GPU and 16 cores of Intel Xeon E5-2670 (Sandy Bridge) 2.60 GHz CPUs.

Our performance is within 90% of the theoretical maximum, as derived in Section 6.1, and denoted by a dashed line. Slight improvement may be possible through register blocking and tuning.

The same code can be easily tuned for other GPUs. Figure 9 shows the performance on a P100 GPU. To compare with the K40c, note that at n = 16, we get about 500 GFlop/s on the P100 *vs.* 180 GFlop/s on the K40c.

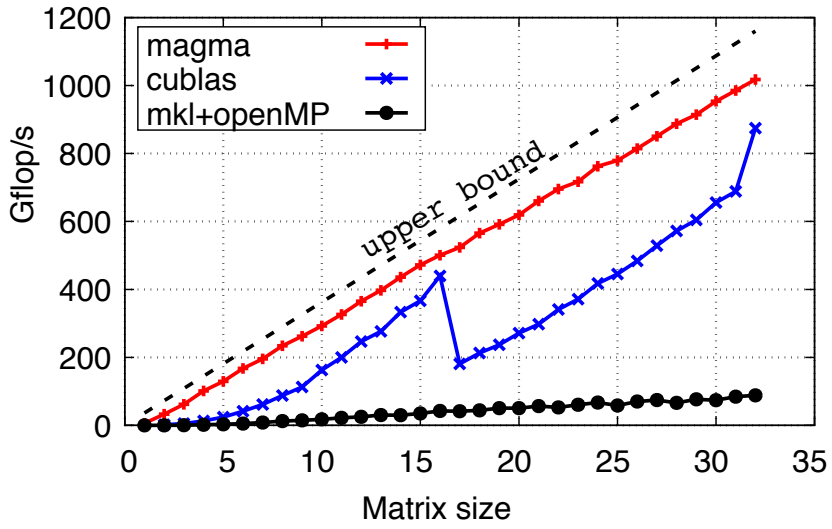

Figure 9: Performance of batched 100,000 DGEMMs on a Tesla P100 GPU.

7.4 Xeon Phi results

Figure 10 gives the performance of batched DGEMMs on KNL without MCDRAM (Left) and with MCDRAM (Right). Note that use of the MCDRAM as the main memory, instead of the

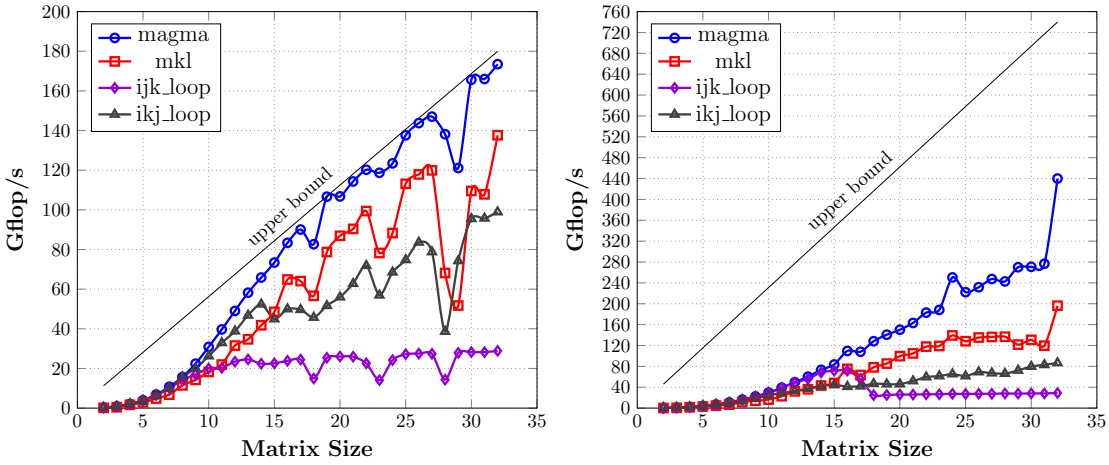

Figure 10: KNL performance of batched DGEMMs without MCDRAM (Left) and with MCDRAM (Right).

DDR4, heavily impacts the performance. Although use of MCDRAM is about twice faster, we end up far from the upper bound due to data in the MCDRAM not being read multiple times, which limits the bandwidth achieved. The SIMD instructions here (AVX512) are 512-bit.

8 Conclusions and future directions

We developed a number of performance optimizations techniques for tensor contractions on new architectures. The contractions targeted are for high-order FEM, and the BLAST code in particular - a high-order FE hydrodynamics research code developed at LLNL. We presented our approach and findings – from the tensor formulation of the numerical FEM algorithms to the definition of tensor interfaces, code generation, design of algorithms, and tuning for performance and portability. We developed a proof-of-concept software that shows that we can achieve performance that is close to optimal. We currently achieve within 90% of a theoretically derived peak on multicore CPUs, GPUs, ARM, and Xeon Phis using on-the-fly tensor reshaping to cast tensor contractions to small but many GEMMs, executed using a batched computing approach, custom-built GEMM kernels for small matrices, and autotuning.

As future work, there are many options that need to be coded and tuned in order to turn the proof-of-concept software developed into production-ready software. The goal is to develop this production-ready HP tensor contractions package and to release it as open source (e.g., through the MAGMA library). Also, it remains to integrate the developments in the BLAST code, and prepare the autotuning in anticipation for easy portability to future supercomputers like Sierra, and new GPU architectures like the Volta with its 3D-stacked memory, expected to further favor tensor computations.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, LLNL release number LLNL-TR-728700.

References

[1] Future directions in tensor-based computation and modeling, May 2009. Workshop. Arlington, Virginia. Technical report published at http://www.cs.cornell.edu/CV/TenWork/FinalReport.pdf.

[2] Workshop on Batched, Reproducible, and Reduced Precision BLAS. `http://bit.ly/Batch-BLAS-2016`, 2016. University of Tennessee, Knoxville, TN, May 18 19.

[3] Workshop on Batched, Reproducible, and Reduced Precision BLAS. `http://bit.ly/Batch-BLAS-2017`, 2017. Georgia Tech, Atlanta, GA, February 23-25.

[4] A. Abdelfattah, M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, J. Falcou, A. Haidar, I. Karlin, Tz. Kolev, I. Masliah, and S. Tomov. Accelerating Tensor Contractions in High-Order FEM with MAGMA Batched. `http://icl.cs.utk.edu/projectsfiles/magma/pubs/52-Tomov-SIAM-CSE2017.pdf`. SIAM Conference on Computer Science and Engineering (SIAM CSE'17) Presentation, Atlanta, GA, February 26-March 3, 2017.

[5] A. Abdelfattah, M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, J. Falcou, A. Haidar, I. Karlin, Tz. Kolev, I. Masliah, and S. Tomov. High-performance tensor contractions for GPUs. *Procedia Computer Science*, 80:108–118, 2016.

[6] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. Performance, design, and autotuning of batched GEMM for GPUs. In *High Performance Computing: 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, pages 21–38. Springer International Publishing, 2016.

[7] M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, J. Falcou, A. Haidar, I. Karlin, T. Kolev, I. Masliah, and S. Tomov. Towards a High-Performance Tensor Algebra Package for Accelerators. http://computing.ornl.gov/workshops/SMC15/presentations/. Smoky Mountains Computational Sciences and Engineering Conference (SMC'15), Gatlinburg, TN, Sep 2015.

[8] N. Bell and J. Hoberock. Thrust: A 2 6. *GPU Computing Gems Jade Edition*, page 359, 2011.

[9] Veselin Dobrev, Tzanio V. Kolev, and Robert N. Rieben. High-order curvilinear finite element methods for lagrangian hydrodynamics. *SIAM J. Scientific Computing*, 34(5), 2012.

[10] Tingxing Dong, Veselin Dobrev, Tzanio Kolev, Robert Rieben, Stanimire Tomov, and Jack Dongarra. A step towards energy efficient computing: Redesigning a hydrodynamic application on CPU-GPU. In *IEEE 28th International Parallel Distributed Processing Symposium (IPDPS)*, 2014.

[11] Jack Dongarra, Iain Duff, Mark Gates, Azzam Haidar, Sven Hammarling, Nicholas J. Higham, Jonathon Hogg, Pedro Valero-Lara, Samuel D. Relton, Stanimire Tomov, and Mawussi Zounon. A proposed API for Batched Basic Linear Algebra Subprograms. MIMS EPrint 2016.25, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, April 2016.

[12] Jing Gong, Stefano Markidis, Erwin Laure, Matthew Otten, Paul Fischer, and Misun Min. Nekbone performance on gpus with openacc and cuda fortran implementations. *The Journal of Supercomputing*, 72(11):4160–4180, 2016.

[13] A. Haidar, A. Abdelfattah, V. Dobrev, I. Karlin, Tz. Kolev, S. Tomov, and J. Dongarra. Tensor Contractions for High-Order FEM on CPUs, GPUs, and KNLs. Smoky Mountains Computational Sciences and Engineering Conference (SMC'16), Gatlinburg, TN, Sep 2016.

[14] Azzam Haidar, Tingxing Dong, Piotr Luszczek, Stanimire Tomov, and Jack Dongarra. Batched matrix computations on hardware accelerators based on GPUs. *IJHPCA*, doi:10.1177/1094342014567546, 02/2015.

[15] Azzam Haidar, Tingxing Dong, Stanimire Tomov, Piotr Luszczek, and Jack Dongarra. A Framework for Batched and GPU-Resident Factorization Algorithms Applied to Block Householder Transformations. In *High Performance Computing*, volume 9137 of *Lecture Notes in Computer Science*, pages 31–47. 2015.

[16] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation. *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 00:981–991.

[17] Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM Rev.*, 51(3):455–500, August 2009.

[18] Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning GEMM for GPUs. In *Proceedings of the 2009 International Conference on Computational Science, ICCS'09*, Baton Roube, LA, May 25-27 2009. Springer.

[19] Wenjing Ma, S. Krishnamoorthy, O. Villay, and K. Kowalski. Acceleration of Streamed Tensor Contraction Expressions on GPGPU-Based Clusters. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pages 207–216, Sept 2010.

[20] Software distribution of MAGMA version 1.6.1. `http://icl.cs.utk.edu/magma/software/`, January 30 2015. Quick reference `http://icl.cs.utk.edu/projectsfiles/magma/pubs/SC14-MAGMA.pdf`.

[21] Stefano Markidis, Jing Gong, Michael Schliephake, Erwin Laure, Alistair Hart, David Henty, Katherine Heisey, and Paul F. Fischer. Openacc acceleration of the nek5000 spectral element code. *IJHPCA*, 29(3):311–319, 2015.

[22] Ian Masliah, Ahmad Abdelfattah, A. Haidar, S. Tomov, Marc Baboulin, J. Falcou, and J. Dongarra. High-performance matrix-matrix multiplications of very small matrices. In *Proceedings of the 22Nd International Conference on Euro-Par 2016: Parallel Processing - Volume 9833*, pages 659–671, Cham, 2016. Springer International Publishing.

[23] David S. Medina, Amik St.-Cyr, and Timothy Warburton. OCCA: A unified approach to multithreading languages. *CoRR*, abs/1403.0968, 2014.

[24] Rajib Nath, Stanimire Tomov, and Jack Dongarra. An Improved MAGMA GEMM For Fermi Graphics Processing Units. *Int. J. High Perform. Comput. Appl.*, 24(4):511–515, November 2010.

[25] Thomas Nelson, Axel Rivera, Prasanna Balaprakash, Mary W. Hall, Paul D. Hovland, Elizabeth R. Jessup, and Boyana Norris. Generating efficient tensor contractions for gpus. Technical report, 2015.

[26] M. Otten, J. Gong, A. Mametjanov, A. Vose, J. Levesque, P. Fischer, and M. Min. An mpi/openacc implementation of a high order electromagnetics solver with gpudirect communication. *International Journal of High Performance Computing Applications*, pages 1–15, 03/2015 2015.

[27] Yang Shi, U. N. Niranjan, Animashree Anandkumar, and Cris Cecka. Tensor contractions with extended BLAS kernels on CPU and GPU. *CoRR*, abs/1606.05696, 2016.

[28] Jaewook Shin, Mary Hall, Jacqueline Chame, Chun Chen, Paul Fischer, and Paul Hovland. Speeding up nek5000 with autotuning and specialization. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 253–262, New York, NY, USA, 2010. ACM.

[29] Edgar Solomonik and James Demmel. Contracting symmetric tensors using fewer multiplications. Technical report, ETH Zürich, 2015.

[30] Edgar Solomonik, Devin Matthews, Jeff Hammond, John Stanton, and James Demmel. A massively parallel tensor contraction framework for coupled-cluster computations. Technical Report UCB/EECS-2014-143, EECS Department, University of California, Berkeley, Aug 2014.

[31] Paul Springer and Paolo Bientinesi. Design of a high-performance gemm-like tensor-tensor multiplication. *CoRR*, abs/1607.00145, 2016.

[32] Kevin Stock, Thomas Henretty, Iyyappa Murugandi, P. Sadayappan, and Robert J. Harrison. Model-driven simd code generation for a multi-resolution tensor kernel. In *IPDPS*, pages 1058–1067. IEEE, 2011.